Stack Applications

Bruce Nan

Applications

- Reverse Data
- Convert Decimal to Binary
- Evaluate Postfix Expressions
- Infix to Postfix Conversion

Quick Introduction

- Stacks are linear data structure.
- All deletions and insertions occur at one end of the stack known as the TOP.
- Data going into the stack first, leaves out last.
- Stacks are also known as LIFO data structures (**L**ast-**I**n, **F**irst-**O**ut).

Basic Stack Operations

- push – Adds an item to the top of a stack.
- pop – Removes an item from the top of the stack and returns it to the user.
- stack top (top, peek) – Copies the top item of the stack and returns it to the user; the item is not removed, hence the stack is not altered.

Additional Notes

- Stacks structures are usually implemented using arrays or linked lists.
- For both implementations, the running time is O(n).
- We will be examining common Stack Applications.

Stack Applications

- Reversing Data: We can use stacks to reverse data. (example: files, strings)
  Very useful for finding palindromes.

Consider the following pseudocode:

```
1)  read (data)
2)  loop (data not finished and stack not full)
          1) push (data)
          2) read (data)
3)  Loop (while stack not Empty)
          1) pop (data)
          2) print (data)
```

Stack Applications

- Converting Decimal to Binary: Consider the following pseudocode

```
1)  Read (number)
2)  Loop (number > 0)
        1) digit = number modulo 2
        2) print (digit)
        3) number = number / 2
// from Data Structures by Gilbert and Forouzan
```

The problem with this code is that it will print the binary number backwards. (ex: 19 becomes 11001000 instead of 00010011.)

To remedy this problem, instead of printing the digit right away, we can push it onto the stack. Then after the number is done being converted, we pop the digit out of the stack and print it.

Evaluation of Expressions

X = a / b - c + d * e - a * c

a = 4, b = c = 2, d = e = 3

Interpretation 1:
((4/2)-2)+(3*3)-(4*2)=0 + 8+9=1

Interpretation 2:
(4/(2-2+3))*(3-4)*2=(4/3)*(-1)*2=-2.66666…

How to generate the machine instructions corresponding to a given expression?

precedence rule + associative rule

Infix to Postfix

Infix	Postfix
A + B	A B +
12 + 60 – 23	12 60 + 23 –
(A + B)*(C – D)	A B + C D – *
A ↑ B * C – D + E/F	A B ↑ C*D – E F/+

Infix to Postfix

- Note that the postfix form an expression does not require parenthesis.
- Consider ‘4+3*5’ and ‘(4+3)*5’. The parenthesis are not needed in the first but they are necessary in the second.
- The postfix forms are:

4+3*5	435*+
(4+3)*5	43+5*

Evaluating Postfix

- Each operator in a postfix expression refers to the previous two operands.
- Each time we read an operand, we push it on a stack.
- When we reach an operator, we pop the two operands from the top of the stack, apply the operator and push the result back on the stack.

Evaluating Postfix

```
Stack s;
while( not end of input ) {
    e = get next element of input
    if( e is an operand )
            s.push( e );
    else {
            op2 = s.pop();
            op1 = s.pop();
            value = result of applying operator ‘e’ to op1 and op2;
            s.push( value );
    }
}
finalresult = s.pop();
```

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	**op1**	**op2**	**value**	**stack**
6				**6**
2				**6,2**
3				**6,2,3**

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	**op1**	**op2**	**value**	**stack**
6				**6**
2				**6,2**
3				**6,2,3**
+	**2**	**3**	**5**	**6,5**
-	**6**	**5**	**1**	**1**

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	**op1**	**op2**	**value**	**stack**
6				**6**
2				**6,2**
3				**6,2,3**
+	**2**	**3**	**5**	**6,5**
-	**6**	**5**	**1**	**1**
3	**6**	**5**	**1**	**1,3**

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
↑	7	2	49	49

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
↑	7	2	49	49
3	7	2	49	49,3

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	op1	op2	value	stack
6				6
2				6,2
3				6,2,3
+	2	3	5	6,5
-	6	5	1	1
3	6	5	1	1,3
8	6	5	1	1,3,8
2	6	5	1	1,3,8,2
/	8	2	4	1,3,4
+	3	4	7	1,7
*	1	7	7	7
2	1	7	7	7,2
↑	7	2	49	49
3	7	2	49	49,3
+	49	3	52	52

Evaluating Postfix

Evaluate 6 2 3 + - 3 8 2 / + * 2 ↑ 3 +

Input	**op1**	**op2**	**value**	**stack**
6				**6**
2				**6,2**
3				**6,2,3**
+	**2**	**3**	**5**	**6,5**
-	**6**	**5**	**1**	**1**
3	**6**	**5**	**1**	**1,3**
8	**6**	**5**	**1**	**1,3,8**
2	**6**	**5**	**1**	**1,3,8,2**
/	**8**	**2**	**4**	**1,3,4**
+	**3**	**4**	**7**	**1,7**
*****	**1**	**7**	**7**	**7**
2	**1**	**7**	**7**	**7,2**
↑	**7**	**2**	**49**	**49**
3	**7**	**2**	**49**	**49,3**
+	**49**	**3**	**52**	**52**

Converting Infix to Postfix

- Consider the infix expressions ‘A+B*C’ and ‘ (A+B)*C’.
- The postfix versions are ‘ABC*+’ and ‘AB+C*’.
- The order of operands in postfix is the same as the infix.
- In scanning from left to right, the operand ‘A’ can be inserted into postfix expression.

Converting Infix to Postfix

- The ‘+’ cannot be inserted until its second operand has been scanned and inserted.
- The ‘+’ has to be stored away until its proper position is found.
- When ‘B’ is seen, it is immediately inserted into the postfix expression.
- Can the ‘+’ be inserted now? In the case of ‘A+B*C’ cannot because * has precedence.

Converting Infix to Postfix

- In case of ‘(A+B)*C’, the closing parenthesis indicates that ‘+’ must be performed first.
- Assume the existence of a function ‘prcd(op1,op2)’ where op1 and op2 are two operators.
- Prcd(op1,op2) returns TRUE if op1 has precedence over op2, FASLE otherwise.

Converting Infix to Postfix

- prcd(‘*’,’+’) is TRUE
- prcd(‘+’,’+’) is TRUE
- prcd(‘+’,’*’) is FALSE
- Here is the algorithm that converts infix expression to its postfix form.
- The infix expression is without parenthesis.

Converting Infix to Postfix

```
Stack s;
While( not end of input ) {
    c = next input character;
    if( c is an operand )
        add c to postfix string;
    else {
        while( !s.empty() && prcd(s.top(),c) ){
            op = s.pop();
            add op to the postfix string;
         }
         s.push( c );
    }
    while( !s.empty() ) {
        op = s.pop();
        add op to postfix string;
    }
```

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
	ABC *	+

Converting Infix to Postfix

- Example: A + B * C

symb	postfix	stack
A	A	
+	A	+
B	AB	+
*	AB	+ *
C	ABC	+ *
	ABC *	+
	ABC * +	

Converting Infix to Postfix

- Handling parenthesis
- When an open parenthesis ‘(‘ is read, it must be pushed on the stack.
- This can be done by setting prcd(op,‘(‘) to be FALSE.
- Also, prcd(‘(‘,op) == FALSE which ensures that an operator after ‘(‘ is pushed on the stack.

Converting Infix to Postfix

- When a ‘)’ is read, all operators up to the first ‘(‘ must be popped and placed in the postfix string.
- To do this, prcd(op,’)’) == TRUE.
- Both the ‘(‘ and the ‘)’ must be discarded: prcd(‘(‘,’)’) == FALSE.
- Need to change line 11 of the algorithm.

Converting Infix to Postfix

```
if( s.empty()  ||  symb != ‘)’ )
      s.push( c );
else
      s.pop(); // discard the ‘(‘
```

prcd(‘(‘, op) = FALSE	for any operator
prcd(op, ‘)’) = FALSE	for any operator other than ‘)’
prcd(op, ‘)’) = TRUE	for any operator other than ‘(‘
prcd(‘)’, op) = error	for any operator.